Random Number Generators:

Introduction for Application Developers

Key material generation is as important to strong cryptosystems as the algorithms used. Weak random number generators (RNGs) have been known to create key material that is guessable by adversaries[1], making the strength of the algorithms irrelevant in cryptographic attacks. This paper, intended for application developers, provides an overview of considerations developers should be making when using RNGs, outlines how RNGs work, and gives guidance for applications needing RNG services.

Why Good RNGs Are Important

Consider an example system using 256-bit AES to protect sensitive communications. Imagine these keys are generated by selecting from a hard-coded list of four possible values. If an attacker knows this list, then he needs at most four guesses to read the sensitive traffic, instead of the 2^256 guesses that should be required to break the AES algorithm. This situation is equivalent to using a weak RNG because an attacker can exhaust the key space.

When designing a security product, there are many situations in which "random" numbers are needed. We can divide these situations into two classes – key material and one-time numbers:

Key Material. Key material includes authentication keys (RSA, DSA, ECDSA), key agreement parameters (DH and ECDH), and encryption keys (3DES, AES, RSA). As the example above indicates, the reason key material needs a good RNG is to prevent guessing. So while an incrementing counter may avoid repeats, an adversary can guess the next output and, thus, guess the key material.

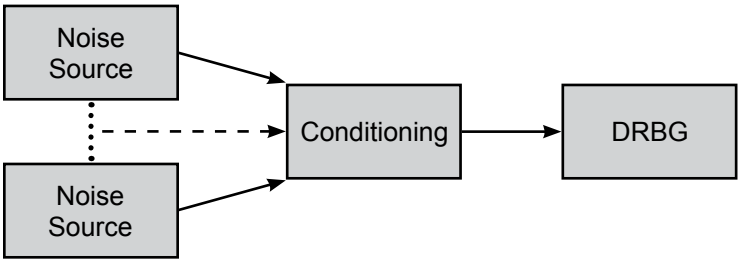
One-Time Numbers. One-time numbers include unique identifiers, session IDs, nonces, and most initialization vectors. With this class, the goal of using an RNG is not to prevent guessing, but to avoid repeats. In these situations, a predictable output, while not recommended, may be acceptable.

What Makes Good RNG Output

Good RNGs produce bytes that are unpredictable. The measurement of this unpredictability is called "entropy." Good RNGs are guaranteed to produce outputs with as much entropy as requested - that is, in a sequence of bits, each bit is equally likely to be zero as one. System services should request as much entropy as the security strength[2] of the algorithm.

How a Good RNG Works

A good RNG usually has three pieces – (1) noise source(s), (2) optional conditioning block, and (3) deterministic random bit generator (DRBG). The RNG should be designed and provided by the hardware or operating system rather than being implemented by application developers.



Where to Get Good RNG Services

Generally, application developers should rely on the platform to provide the RNG. The platform RNG should have both its noise sources and DRBG validated to provide application developers assurance as to the quality of the random bits generated. The flowchart on the following page gives recommendations for developers based on the validation status of the platform.

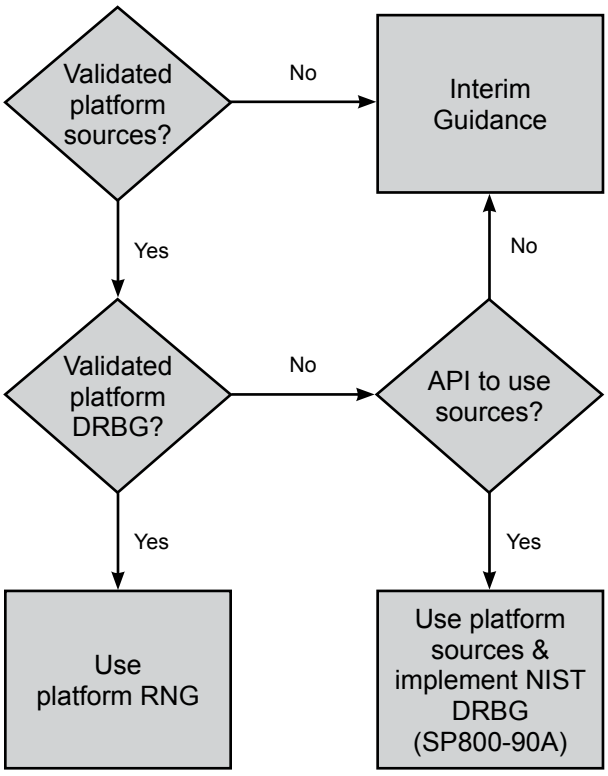
Interim Guidance

As many platforms do not have validated sources or DRBGs, the following per-platform guidance should be followed until validation occurs:

[1] Durumeric, Z., Halderman, J., Heninger, N., Wustrow, E. "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices." In *Proc. 21st USENIX Security Symposium*, Aug 2012. Rev. 2.

[2] Security Strength is not necessarily the size of the key. NIST SP800-57 describes security strength and gives the strengths for various algorithms.

Validation Status Flowchart



- *Microsoft Windows Desktop Applications (since Windows XP, Windows Server 2003):* Application developers should use the `CryptGenRandom` function for cryptographic RNG services. **http://msdn.microsoft.com/en-us/library/aa923614.aspx**

- *Microsoft Windows Store Apps (since Windows 8, Windows Server 2012):* App developers should use the `CryptographicBuffer.GenerateRandom` method for cryptographic RNG services. **http://msdn.microsoft.com/en-us/library/windows/apps/windows.security.cryptography.cryptographicbuffer.generaterandom**

- *Blackberry (since BlackBerry 10):* Application developers should use the `RngGetBytes` function with either the ANSI X9.31 AES RNG or one of the SP800-90 DRBGs with the appropriate security strength for cryptographic RNG services. **http://developer.blackberry.com/native/reference/core/com.qnx.doc.crypto.lib_ref/topic/hu_rnggetbytes.html**

- *Apple iOS (since iOS 2.0):* Application developers should use the `SecRandomCopyBytes` function with the `kSecRandomDefault` random number generator for RNG services. **https://developer.apple.com/library/ios/docuemntation/Security/Reference/RandomizationReference/Reference/reference.html**

- *Unix-like Platforms (e.g. Linux, Android, and Mac OS X)*: Application developers should use the `fread` function to read random bytes from `/dev/random` for cryptographic RNG services.
  Because `/dev/random` is a blocking device, `/dev/random` may cause unacceptable delays, in which case application developers may prefer to implement a DRBG using `/dev/random` as a conditioned seed. Application developers should use the “Random Number Generators: Introduction for Operating System Developers” guidance in developing this solution.
  If `/dev/random` still produces unacceptable delays, developers should use `/dev/urandom` which is a non-blocking device, but only with a number of additional assurances:
  - The entropy pool used by `/dev/urandom` must be saved between reboots.
  - The Linux operating system must have estimated that the entropy pool contained the appropriate security strength entropy at some point before calling `/dev/urandom`. The current pool estimate can be read from `/proc/sys/kernel/random/entropy_avail`.
  - At most 2^80 bytes may be read from `/dev/urandom` before the developer must ensure that new entropy was added to the pool.